A Compressed Breadth-First Search for Satisfiability

DoRon B. Motter and Igor L. Markov

Department of EECS, University of Michigan, 1301 Beal Ave, Ann Arbor, MI 48109-2122
{dmotter, imarkov}@eecs.umich.edu

Abstract. Leading algorithms for Boolean satisfiability (SAT) are based on either a depth-first tree traversal of the search space (the DLL procedure [6]) or resolution (the DP procedure [7]). In this work we introduce a variant of Breadth-First Search (BFS) based on the ability of Zero-Suppressed Binary Decision Diagrams (ZDDs) to compactly represent sparse or structured collections of subsets. While a BFS may require an exponential amount of memory, our new algorithm performs BFS directly with an implicit representation and achieves unconventional reductions in the search space.
We empirically evaluate our implementation on classical SAT instances difficult for DLL/DP solvers. Our main result is the empirical $\Theta(n^4)$ runtime for hole-n instances, on which DLL solvers require exponential time.

1 Introduction

Efficient methods to solve SAT instances have widespread applications in practice and have been the focus of much recent research [16, 17]. Even with the many advances, a number of practical and constructed instances remain difficult to solve. This is primarily due to the size of solution spaces to be searched.

Independently from SAT, [9] explored Lempel-Ziv compression in exhaustive search applications such as game-playing and achieved memory reductions by a small constant factor. Text searches in properly indexed databases compressed using a Burrows-Wheeler scheme were proposed in [8]. However, these works enable only a limited set of operations on compressed data, and the asymptotic compression ratios are too small to change the difficulty of search for satisfiability. A different type of compression is demonstrated by Reduced Ordered Binary Decision Diagrams (ROBDDs) and Zero-Suppressed Binary Decision Diagrams (ZDDs) [13]. In general, Binary Decision Diagrams implicitly represent combinatorial objects by the set of paths in a directed acyclic graph. Complexity of algorithms that operate on BDDs is often polynomial in the size of the BDD. Therefore, by reducing the size of BDDs, one increases the efficiency of algorithms. There have been several efforts to leverage the power of BDDs and ZDDs as a mechanism for improving the efficiency of SAT solvers. [4] [5] used ZDDs to store a clause database and perform DP on implicit representations. [2] used ZDDs to compress the containers used in DLL. Another method is to iteratively construct the BDD corresponding to a given CNF formula [19]. In general this method has a different class of tractable instances than DP [11]. Creating this BDD is known to require exponential time for several instances [11], [19]. For random 3-SAT instances, this method also gives slightly different behavior than DP/DLL solvers [12].

The goal of our work is to use ZDDs in an entirely new context, namely to make Breadth-First Search (BFS) practical. The primary disadvantage of BFS — exponential memory requirement — often arises as a consequence of explicit state representations (queues, priority queues). As demonstrated by our new algorithm Cassatt, integrating BFS with a compressed data structure significantly extends its power. On classical, hard SAT benchmarks, our implementation achieves asymptotic speed-ups over published DLL solvers. To the best of our knowledge, *this is the first work in published literature to propose a compressed BFS as a method for solving the SAT decision problem.* While Cassatt does not return a satisfying solution if it exists, any such SAT oracle can be used to find satisfying solutions with at most $|Vars|$ calls to this oracle.

The remaining part of the paper is organized as follows. Section 2 covers the background necessary to describe the Cassatt algorithm. A motivating example for our work is shown in Section 3. In Section 4 we discuss the implicit representation used in Cassatt, and the algorithm itself is described in Section 5. Section 6 presents our experimental results. Conclusions and directions of our ongoing work are described in Section 7.

2 Background

A *partial truth assignment* to a set of Boolean variables V is a mapping $t : V \rightarrow \{0, 1, *\}$. For some variable $v \in V$, if $t(v) = 1$ then the literal v is said to be "true" while the literal $\bar{v}$ is said to "false" (and vice versa if $t(v) = 0$). If $t(v) = *$, v and $\bar{v}$ are said not to be assigned values. Let a *clause* denote a set of literals. A clause is *satisfied* by a truth assignment t iff at least one of its literals is true under t. A clause is said to be *violated* by a truth assignment t if all of its literals are false under t. A Boolean formula in conjunctive normal form (CNF) can be represented by a set C of clauses.

2.1 Clause Partitions

For a given Boolean formula in CNF, a partial truth assignment t is said to be *invalid* if it violates any clauses. Otherwise, it is *valid*. The implicit state representation used in the Cassatt algorithm relies on the partition of clauses implied by a given valid partial truth assignment. Each clause c must fall into exactly one of the three categories shown in Figure 1.

We are going to compactly represent multiple partial truth assignments that share the same set of assigned variables. Such partial truth assignments appear in the process of performing a BFS for satisfiability to a given depth. Note that simply knowing which variables have been assigned is enough to determine which clauses are *unassigned* or not *activated*. The actual truth values assigned differentiate *satisfied* clauses from *open* clauses. Therefore, if the assigned variables are known, we will store the set of *open* clauses rather than the actual partial truth assignment. While the latter may be impossible to recover, we will show in the following sections that the set of *open* clauses contains enough information to perform a BFS.

Another important observation is that only clauses which are "cut" by the vertical line in Figure 1 have the potential to be *open* clauses. The number of such cut clauses

- **Unassigned Clauses**:
  Clauses whose literals are unassigned.
- **Satisfied Clauses**:
  Clauses which have at least one literal satisfied.
- **Open Clauses**:
  Clauses which have at least one, but not all of their literals assigned, and are not satisfied.

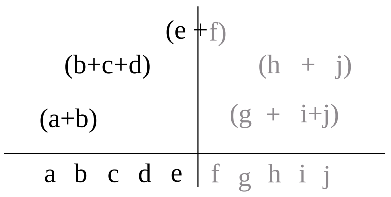

Fig. 1. The three clause partitions entailed by a valid partial truth assignment. The vertical line separates assigned variables (a through e) from unassigned variables (f through j, grayed out). We distinguish (i) clauses to the right of the vertical line, (ii) clauses to the left of the vertical line, and (iii) clauses cut by the vertical line.

does not depend on the actual assigned values, but depends on the order in which variables are processed. This observation leads to the use of MINCE [1] as a heuristic for variable ordering since it reduces *cutwidth*. MINCE was originally proposed as a variable ordering heuristic to complement nearly any SAT solver or BDD-based application. MINCE treats a given CNF as a (undirected) hypergraph and applies recursive balanced min-cut hypergraph bisection in order to produce an ordering of hypergraph vertices. This algorithm is empirically known to achieve small values of total hyperedge span. Intuitively, this algorithm tries to minimize cutwidth in many places, and as such is an ideal complement to the Cassatt algorithm. Its runtime is estimated to be $O((V+C)log^2V)$, where V is the number of variables and C is the number of clauses in a given CNF. By reducing the *cutwidth* in Cassatt, we obtain an exponential reduction in the number of possible sets of open clauses. However, storing each set explicitly is still often intractable. To efficiently store this collection of sets, we use ZDDs.

2.2 Zero-Suppressed Binary Decision Diagrams

In Cassatt we use ZDDs to attempt to represent a collection of N objects often in fewer than N bits. In the best case, N objects are stored in $O(p(logN))$ space where $p(n)$ is some polynomial. The compression in a BDD or ZDD comes from the fact that objects are represented by paths in a directed acyclic graph (DAG), which may have exponentially more paths than vertices or edges. Set operations are performed as graph traversals over the DAG. Zero-Suppressed Binary Decision Diagrams (ZDDs) are a variant of BDDs which are suited to storing collections of sets. An excellent tutorial on ZDDs is available at [14].

A ZDD is defined as a directed acyclic graph (DAG) where each node has a unique label, an integer index, and two outgoing edges which connect to what we will call *T-Child* and *E-child*. Because of this we can represent each node X as a 3-tuple $X\langle n, X_T, X_E\rangle$ where n is the index of the node X, X_T is its *T-Child*, and X_E is its *E-Child*. Each path in the DAG ends in one of two special nodes, the **0** node and the **1** node. These nodes have no successors. In addition, there is a single root node. When we use a ZDD we will in reality keep a reference to the root node. The semantics of a ZDD can be defined recursively by defining the semantics of a given node.

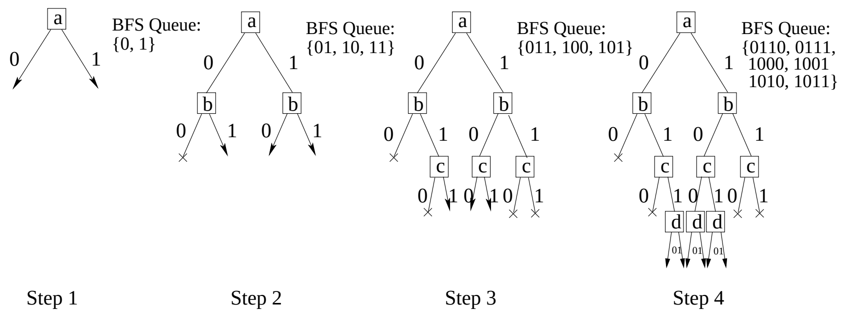
BFS Queue: {0, 1}
BFS Queue: {01, 10, 11}
BFS Queue: {011, 100, 101}
BFS Queue: {0110, 0111, 1000, 1001 1010, 1011}
Step 1
Step 2
Step 3
Step 4

Fig. 2. Steps Taken by Breadth-First Search

A ZDD can be used to encode a collection of sets by encoding its characteristic function. We can evaluate a function represented by a ZDD by traversing the DAG beginning at the root node. At each node X, if the variable corresponding to the index of X is true, we select the *T-Child*. Otherwise we select with the *E-Child*. Eventually we will reach either **0** or **1**, indicating the value of the function on this input. We augment this with the *Zero-Suppression Rule*: we may eliminate nodes whose $T-Child$ is **0**. With these standard rules, **0** represents the empty collection of sets, while **1** represents the collection consisting of only the empty set. ZDDs interpreted this way have a standard set of operations based on recursive definitions [13], including the union and intersection of two collections of sets, for example.

3 A Motivating Example

$$\underbrace{(a+b)}_{1}\underbrace{(\overline{b}+c)}_{2}\underbrace{(d+e)}_{3}\underbrace{(\overline{a}+\overline{b}+\overline{c})}_{4}\underbrace{(c+\overline{d}+e)}_{5}$$

3.1 Steps Taken by Breadth-First Search

In general, BFS expands nodes in its queue until reaching a violated clause, at which point the search space is pruned. As a result, the number of nodes grows quickly.

The initial pass of the BFS considers the first variable a. Both $a=0$ and $a=1$ are valid partial truth assignments since neither violates any clauses. BFS continues by enqueueing both of these partial truth assignments. In Figure 2, the contents of the BFS Queue are listed as bit-strings. At Step 2, the BFS considers both possible values for b. Because of the clause $(a+b)$, BFS determines that $a=0$, $b=0$ is not a valid partial truth assignment. The remaining three partial truth assignments are valid, and BFS enqueues them. At Step 3, because of the clause $(\overline{a}+\overline{b}+\overline{c})$ we know that a, b, and c cannot all be true. The search space is pruned further because $(\overline{b}+c)$ removes all branches involving $b=1, c=0$. At Step 4, the state space doubles.

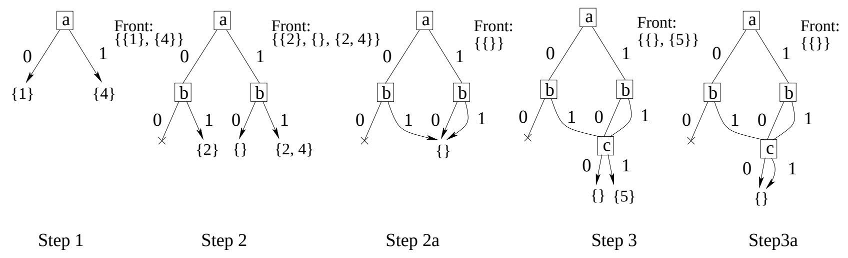

Fig. 3. Steps Taken by Cassatt

3.2 Steps Taken by Cassatt

The Cassatt algorithm performs a similar search, however partial truth assignments are not stored explicitly. As a result, if two partial truth assignments correspond to the same set of open clauses then only one of these will be stored. While the algorithm is described in Section 5, here we demonstrate possible reductions in the search space. We will call Cassatt's collection of sets of open clauses the *front*.

Variable a appears in two clauses: clause 1, $(a+b)$ and clause 4, $(\overline{a}+\overline{b}+\overline{c})$. If the variable a is assigned $a=1$, then clause 1 becomes satisfied while making clause 4 *open*. If $a=0$, then clause 1 becomes *open*. Because we cannot yet determine which of these assignments (if any) will lead to to a satisfying truth assignment, it is necessary to store both branches. The *front* shown in Figure 3, Step 1, is updated to contain the two sets of open clauses which correspond to the two possible valid partial truth assignments. Variable b appears in three clauses: clause 1, $(a+b)$, clause 2, $(\overline{b}+c)$, and clause 4, $(\overline{a}+\overline{b}+\overline{c})$. These clauses are our new set of activated clauses. For the first set, $\{1\}$, we consider both possible truth assignments, $b=0$ and $b=1$. Assigning $b=0$ satisfies clauses 2 and 4. However, it does not satisfy clause 1. We also see that b corresponds to the end literal for clause 1. If we do not choose the value of b which satisfies this clause, it will never be satisfied. As a result we cannot include this branch in the new *front*. Note that realizing this only depends on noticing that b corresponds to the end literal for some open clause. For $b=1$ we see that clause 1 is satisfied, but clauses 2 and 4 are not yet satisfied. So, $\{2,4\}$ should be in the *front* after this step is completed.

A similar analysis can be performed for the remaining set in the *front*, $\{4\}$. In Cassatt it is not necessary to consider these sets in succession; ZDD algorithms allow us to perform these operations on multiple sets simultaneously. These sets of clauses, listed in the *front* shown in Step 2, are all the state information Cassatt maintains. It is here that this algorithm begins to differ substantially from a naive BFS. One of the sets in the *front* is the empty set. This means that there is some assignment of variables which could satisfy every clause *in which any of these variables appear*. As a result, there is no need to examine any other truth assignment; the partial truth assignment which leaves no open clauses is clearly and provably the best choice. As a result of storing sets of open clauses, Cassatt prunes the search space and only considers the single node.

Whereas the traditional BFS needs to consider three alternatives at this point, Cassatt, by storing open clauses, deduces immediately that one of these is provably superior. As a result, only a single node is expanded. As seen in Step 2a, this effectively restructures the search space so that several of the possible truth assignments are subsumed into a single branch.

As Cassatt processes the third variable, c, it needs to only consider its effects on the single, empty subset of open clauses. Since each new variable activates more clauses, this empty subset once again, can potentially grow. Again, one possible variable choice satisfies all activated clauses. This is shown in Step 3. As a result, Cassatt can once again subsume multiple truth assignments into a single branch as indicated in Step 3a.

4 Advantages of Storing Open Clauses

If two partial truth assignments create the exact same set of open clauses, then clearly space can be saved by only storing one of these subsets. This also cuts the search space. Moreover, if a stored set of clauses A is a proper subset of another set of clauses B, then the partial truth assignment corresponding to B is *sub-optimal* and can be discarded. This is so because *every* clause which is affected by any partial truth assignment thus far (every *activated* clause) has been considered when producing A and B. As a result, there are no consequences to choosing the truth assignment which produces A over the truth assignment which produces B. Every satisfying truth assignment based on the partial truth assignment giving B corresponds to a satisfying truth assignment based on A. These subsumption rules are naturally addressed by the ZDD data structure, which appears both appropriate and effective as a compact data structure.

We use three ZDD operations originally introduced in [4]. The *subsumed difference* X_A/X_B is defined as the collection of all sets in X_A which are not subsumed by some set in X_B. Based on the subsumed difference operator, it is possible to define an operator $NoSub(X)$ as the collection of all sets in X which are not subsumed by some other set in X. Finally, the *subsumption-free union* $X_A \bigcup_S X_B$ is defined as the collection of sets in $X_A \bigcup X_B$ from which all subsumed sets have been removed. For the sake of brevity, we will not repeat the full, recursive definitions of these operators here, but only point out that the operators introduced in [4] were shown for slightly different ZDD semantics — to encode collections of clauses. Here we use the original ZDD semantics proposed by Minato [13] — to encode collections of subsets. However, the definitions of operators themselves are unaffected. These operators provide a mechanism for maintaining a subsumption-free collection of sets.

ZDDs are known to often achieve good compression when storing sparse collections. In general a collection of sets may or may not compress well when represented by ZDDs. However in Cassatt our representation can be pruned by removing sets which are subsumed by another set. Intuitively, this leads to a sparser representation for two reasons: sets with small numbers of elements have a greater chance of subsuming larger sets in this collection, and each subsumption reduces the number of sets which need to be stored. By using operators which eliminate subsumed sets, then, we hope to improve the chance that our given collection will compress well. In Cassatt this corresponds to improved memory utilization and runtime.

A natural question to ask is whether the number of subsets to be stored (the *front*) has an upper bound. An easy bound based on the *cutwidth* is obtained by noting that for c clauses, the maximum number of incomparable subsets is exactly the size of the maximal anti-chain of the partially ordered set $\mathbf{2}^c$.

Theorem 4.1 [10, p. 444] The size of the maximal anti-chain of $\mathbf{2}^c$ is given by $\binom{c}{\lfloor \frac{c}{2} \rfloor}$.

This simple upper bound does not take into account the details of Cassatt. Doing this may yield a substantially tighter bound.

It should be clear from the example that simply by storing sets of open clauses, Cassatt potentially searches a much smaller space than a traditional BFS. In general, it cannot fully avoid the explosion in space. Although there are potentially many more combinations of clauses than there are variables, Cassatt cannot examine more nodes than a straightforward BFS. At each stage, Cassatt has the potential to reduce the number of nodes searched. Even if no reduction is ever possible, Cassatt will only examine as many nodes as a traditional BFS.

5 Compressed Breadth-First Search

The Cassatt algorithm implements a breadth-first search based on keeping track of sets of open clauses from the given CNF formula. The collection of these sets represents the state of the algorithm. We refer to this collection of sets as the *front*. The algorithm advances this *front* just as a normal breadth-first algorithm searches successively deeper in the search space. The critical issue is how the *front* can be advanced in a breadth-first manner and determine satisfiability of a CNF formula. To explain this, we will first be precise about what should happen based on combining a single set of open clauses with a truth assignment to some variable v. We will then show how this can be done to the collection of sets by using standard operations on ZDDs.

Given a set of open clauses corresponding to a valid partial truth assignment t, and some assignment to a single variable $v \leftarrow r$, there are two main steps which must be explained. First, Cassatt determines if the combination of t with $v \leftarrow r$ produces a new valid partial truth assignment. Then, Cassatt must produce the new set of open clauses corresponding to the combination of t with $v \leftarrow r$. Since the *front* corresponds to the collection of valid partial truth assignments, then if this *front* is ever empty (equal to $\mathbf{0}$), then the formula is unsatisfiable. If Cassatt processes all variables without the *front* becoming empty then no clauses will be activated, but unsatisfied. Thus if the formula is satisfiable, the *front* will contain only the empty set (equal to $\mathbf{1}$) at the end of the search.

5.1 Detecting Violated Clauses

With respect to the variable ordering used, literals within a clause are divided into three categories: *beginning*, *middle*, and *end*. The *beginning* literal is processed first in the variable ordering while the *end* literal is processed last. The remaining literals are called *middle* literals. A nontrivial clause may have any (nonnegative) number of middle literals. However, it is useful to guarantee the existence of a *beginning* and an *end* literal.

If a clause does not have at least 2 literals, then it forces the truth assignment of a variable, and the formula can be simplified. This is done as a preprocessing step in Cassatt. Cassatt knows in advance which literal is the *end* literal for a clause, and can use this information to detect violated clauses. Recall that a violated clause occurs when all the variables which appear within a clause have been assigned a truth value, yet the clause remains unsatisfied. Thus only the *end* literal has the potential to create a violated clause. The value which does not satisfy this clause produces a conflict or violated clause. This is the case since the sets stored in the *front* contain only unsatisfied clauses. After processing the *end* variable, no assignment to the remaining variables can affect satisfiability.

Determining if the combination of t and $v \leftarrow r$ is a valid partial truth assignment is thus equivalent to considering if v corresponds to an *end* literal for some clause c in the set corresponding to t. If the assignment $v \leftarrow r$ causes this literal to be $false$ then all literals in c are false, and combining t with $v \leftarrow r$ is not valid.

5.2 Determining the New Set of Open Clauses

Given a set of clauses S corresponding to some partial truth assignment and an assignment to a single variable v, all clauses in S which do not contain any literal corresponding to v's should, by default, propagate to the new set S'. This is true, since any assignment to v does not affect these clauses in any way. Consider a clause not in S containing v as its *beginning* variable. If the truth assignment given to v causes this clause to be satisfied, then it should *not* be added to S'. Only if the truth assignment causes the clause to remain unsatisfied does it become an open clause and should be added to S'. Finally if a clause in S contains v as one of its *middle* variables then the literal corresponding to this variable must again be examined. If the truth assignment given to v causes this clause to become satisfied, then it should not be propagated to S'. Otherwise, it should propagate from S to S'.

Table 1. New-Subset Rules

	Beginning	Middle	End	None
Satisfied	Impossible	No Action	No Action	No Action
Open	Impossible	**if**$(t(l) = 0)$ $S' \leftarrow S' \bigcup C$	No Action	$S' \leftarrow S' \bigcup C$
Unassigned	**if**$(t(l) = 0)$ $S' \leftarrow S' \bigcup C$	Impossible	Impossible	No Action

These rules are summarized in table 1. For a given clause and an assignment to some variable, the action to be taken can be determined by where the literal appears in the clause (column) and the current status of this clause (row). Here, “No Action” means that the clause should not be added to S'.

It should be noted that when an *end* variable appears in an open clause, no action is necessary only with regard to S' (it does not propagate into S'). If the assignment does not satisfy this clause, then the partial truth assignment is invalid. However, given that a partial truth assignment is valid then these clauses must become satisfied by that assignment.

5.3 Performing Multiple Operations via ZDDs

Cassatt’s behavior for a single set was described in Table 1. However, the operations described here can be reformulated as operations on the entire collection of sets. By simply iterating over each set of open clauses and performing the operations in Table 1, one could achieve a *correct*, if inefficient implementation. However, it is possible to represent the combined effect on all sets in terms of ZDD operations. In order to create an *efficient* implementation, we perform these ZDD operations on the entire *front* rather than iterating over each set.

To illustrate this, consider how some of the clauses in each subset can be violated. Recall that if a clause c is violated, then the set containing c cannot lead to satisfiability and must be removed from the *front*. The same violated clauses will appear in many sets, and *each* set in the *front* containing any violated clause must be removed. Instead of iterating over each set, we intersect the collection of all *possible* sets without any violated clause with the *front*; all sets containing any violated clause will be removed. Such a collection can be formulated using ZDDs.

More completely, a truth assignment t to a single variable v has the following effects:

- It *violates* some clauses. Let $U_{v,t}$ be the set of all clauses c such that the *end* literal of c corresponds to v and is false under t. Then $U_{v,t}$ is the set of clauses which are violated by this variable assignment.
- It *satisfies* some clauses. Let $S_{v,t}$ be the set of all clauses c such that the literal of c corresponding to v is true under t. If these clauses were not yet satisfied, then they become satisfied by this assignment.
- It *opens* some clauses. Let $A_{v,t}$ be the set of all clauses c such that the *beginning* literal of c corresponds to v and is false under t. Then $A_{v,t}$ is the set of clauses which have just been activated, but remain unsatisfied.

Note that each of these sets depends only on the particular truth assignment to v. With each of these sets of clauses, an appropriate action can be taken on the entire *front*.

5.4 Updating Based on Newly Violated Clauses

Given a set of clauses $U_{v,t}$ which have just been violated by a truth assignment, all subsets which have one or more of these clauses must be removed from the *front*.

Each subset containing any of these clauses cannot yield satisfiability. We can build the collection of all sets which do not contain any of these clauses. This new collection of sets will have a very compact representation when using ZDDs. Let $U'_{v,t}$ denote the collection of all possible sets of clauses which do not contain any elements from $U_{v,t}$. That is, $S \in U'_{v,t} \rightarrow S \bigcap U_{v,t} = \emptyset$. The ZDD corresponding to $U'_{v,t}$ is shown in Figure 4.

In Figure 4, the dashed arrow represents a node's *E-Child* while a solid arrow represents a node's *T-Child*. Because of the Zero-Suppression rule, any node which does not appear in the ZDD is understood not to appear in any set in the collection. As a result, to create a ZDD without clauses in $U_{v,t}$, no element in $U_{v,t}$ should appear as a node in the ZDD for $U'_{v,t}$. Because we look at a finite set of clauses C, we wish to represent the *remaining clauses* $C \setminus U_{v,t}$ as *don't care* nodes in the ZDD. To form the *don't care* nodes, we simply create a ZDD node with both its *T-child* and its *E-child* pointing to the same successor. The resulting ZDD has the form shown in Figure 4a. When we take *front* $\leftarrow$ *front* $\bigcap U'_{v,t}$ we will get all subsets which do not contain any clauses in $U_{v,t}$. As a result, the *front* will be pruned of all partial truth which become invalid as a result of this assignment.

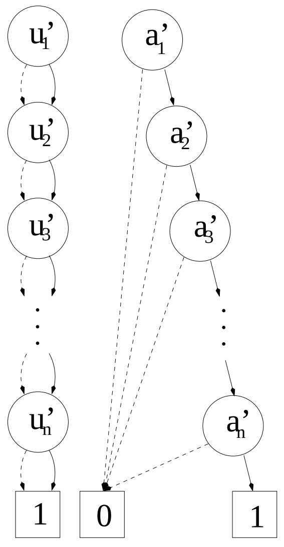

Fig. 4: (a) The $U'_{v,t}$ ZDD. (b) The $A'_{v,t}$ ZDD

Instead of including all clauses in $C \setminus U_{v,t}$, we can further reduce this operation by including only clauses within the *cut*. Including nodes to preserve clauses outside the *cut* is superfluous. The ZDD representing $U'_{v,t}$ can thus be created containing $O(cutwidth)$ nodes in the worst case.

5.5 Updating Based On Newly Satisfied Clauses

Consider a clause which has just been satisfied and appears somewhere in the *front*. Since the *front* consists of sets of activated, unsatisfied clause, then it must have been open (unsatisfied) under some partial truth assignment. However, this occurrence has just been satisfied. As a result, *every* occurrence of a satisfied clause can simply be removed entirely from the *front*.

```
for each $c \in S_{v,t}$ do
    $Temp_0 \leftarrow$ Subset0(Front, c)
    $Temp_1 \leftarrow$ Subset1(Front, c)
    Front $\leftarrow Temp_0 \bigcup Temp_1$
```

Fig. 5: Pseudocode for $\exists$ abstraction

Among other operations, the standard operations on ZDDs include cofactoring, or *Subset0* and *Subset1*. *Subset0(Z, a)* creates the collection of all sets in the ZDD *Z* without a given element *a*, while *Subset1(Z, a)* creates the collection of all sets in a ZDD *Z* which did contain a given element *a*. However, the element *a* is not present in any of the ZDDs returned by either *Subset0* or *Subset1*. As a result, these operations can be used to eliminate a given element from the *front*. This idea is illustrated in the pseudocode in Figure 5.

The combined effect of these operations on a ZDD is called *existential abstraction* [14] and can be implemented with a single operation on the ZDD [15].

5.6 Updating Newly Opened Clauses

Consider a set of newly opened clauses $A_{v,t}$. Each set in the *front* should now also include every element in $A_{v,t}$. We use the ZDD *product* to form the new *front* by letting $A'_{v,t} \equiv \{A_{v,t}\}$ and finding $front \times A'_{v,t}$.

When we build the ZDD $A'_{v,t}$, only those clauses in $A_{v,t}$ need to appear as nodes in $A'_{v,t}$. This is because of the Zero-Suppression rule: nodes not appearing in the ZDD $A'_{v,t}$ must be 0. The ZDD for $A'_{v,t}$ is simple as it only contains a single set. The form of all such ZDDs is shown in Figure 4b. When the ZDD product operation is used, $front \times A'_{v,t}$ will form the new *front*.

In many cases, a shortcut can be taken. By properly numbering clauses with respect to a given variable ordering, at each step we can ensure that each clause index is lower than the minimum index of the *front*. In this case, all clauses should be placed above the current *front*. The *front* will essentially remain unchanged, however new nodes in the form of Figure 4b will be added parenting the root node of the *front*. In this case, the overhead of this operation will be linear in the number of opened clauses.

```
CASSATT($Vars$, $Clauses$)
  Front ← 1
  for each $v \in Vars$ do
     Front$^\dagger$ ← Front
     Form sets $U_{v,0}, U_{v,1}, S_{v,0}, S_{v,1}, A_{v,0}, A_{v,1}$
     Build the ZDDs $U'_{v,0}, U'_{v,1}$
        Front ← Front $\bigcap U'_{v,1}$
        Front$^\dagger$ ← Front$^\dagger$ $\bigcap U'_{v,0}$
        Front ← ∃Abstract(Front, $S_{v,1}$)
        Front$^\dagger$ ← ∃Abstract(Front$^\dagger$, $S_{v,0}$)
     Build the ZDDs $A'_{v,0}, A'_{v,1}$
        Front ← Front $\times A'_{v,1}$
        Front$^\dagger$ ← Front$^\dagger$ $\times A'_{v,0}$
     Front ← Front $\bigcup_S$ Front$^\dagger$
  if Front == 0 then
     return Unsatisfiable
  if Front == 1 then
     return Satisfiable
```

Fig. 6: Pseudocode for the Cassatt Algorithm

5.7 Compressed BFS Pseudocode

Now that we have shown how the *front* can be modified to include the effects of a single variable assignment, the operation of the complete algorithm should be fairly clear. For each variable *v*, we will copy the *front* at a given step, then modify one copy to reflect assigning $v = 1$. We will modify the other copy to reflect assigning $v = 0$. The new *front* will be the union with subsumption of these two.

ZDD nodes are in reality managed via reference counts. When a ZDD (or some of its nodes) are no longer needed, we decrement this reference count. When we copy a ZDD, we simply increment the reference count of the root node. Thus, copying the *front* in Cassatt takes only constant overhead. The pseudocode of the Cassatt algorithm is shown in Figure 6.

Here, we order the steps of the algorithm according to the MINCE ordering, applied as a separate step before Cassatt begins. We initially set the *front* to the collection containing the empty set. This is consistent, because trivially there are no open clauses yet. Recall that after all variables are processed, either the *front* will contain only the empty set (be **1**), or will be the empty collection of sets (be **0**).

6 Empirical Validation

We implemented Cassatt in C++ using the CUDD package [18]. We also used an *existential abstraction* routine from the Extra library [15]. For these results, we disabled reordering and garbage collection. These tests were performed on an AMD Athlon 1.2GHz machine, with 1024MB of 200MHz DDR RAM running Debian Linux. We also include performance of Chaff [16] and GRASP [17], two leading DLL-based SAT solvers. We also include results of ZRes, a solver which performs the DP procedure [7] while compressing clauses with a ZDD. All solvers were used with their default configurations, except ZRes which requires a special switch when solving hole-n instances. All runs were set to time out after 500s.

We ran two sets of benchmarks which are considered to be difficult for traditional SAT solvers [3]. The *hole-n* family of benchmarks are come from the pigeonhole principle. Given $n+1$ pigeons and n holes, the pigeonhole principle implies that least 2 pigeons must live in the same hole. The *hole-n* benchmarks are a CNF encoding of the negation of this principle, and are unsatisfiable as they place at most 1 pigeon in each hole. The number of clauses in this family of benchmarks grows as $\Theta(n^3)$ while the number of variables grows as $\Theta(n^2)$. For hole-50, there are 2550 variables and 63801 clauses. The *Urquhart* benchmarks are a randomly generated family relating to a class of problems based on expander graphs [20]. Both families have been used to prove lower bounds for runtime of classical DP and DLL algorithms and contain only unsatisfiable instances. [3] shows that any DLL or resolution procedure requires $\Omega(2^{n/20})$ time for hole-n instances. Figure 7 empirically demonstrates that Cassatt requires $\Theta(n^4)$ time for these instances. This does not include time used to generate the MINCE variable ordering or I/O time.

The runtimes of the four solvers tested on instances from hole-n are graphed in Figure 7a. The smallest instance we consider is hole-5; Cassatt efficiently proves unsatisfiability of hole-50 in under 14 seconds. Figure 7b plots the runtimes of Cassatt on hole-n vs. n for $n < 50$ on a log-log scale. We also include a plot of cn^4 with a constant set to clearly show the relationship with the empirical data (on the log-log scale, this effectively translates the plot vertically).

The Urquhart-n instances have some degree of randomness. To help compensate for this, we tested 10 different randomly generated instances for each n, and took the average runtime of those instances which completed. Memory appears to be the limiting factor for Cassatt, rather than runtime; it cannot solve one Urq-8 instance, while quickly solving others. With respect to the DIMACS benchmark suite, Cassatt efficiently solves many families. Most *aim* benchmarks are solved, as well as *pret*, and *dubois* instances. However, DLL solvers outperform Cassatt on many benchmarks in this suite.

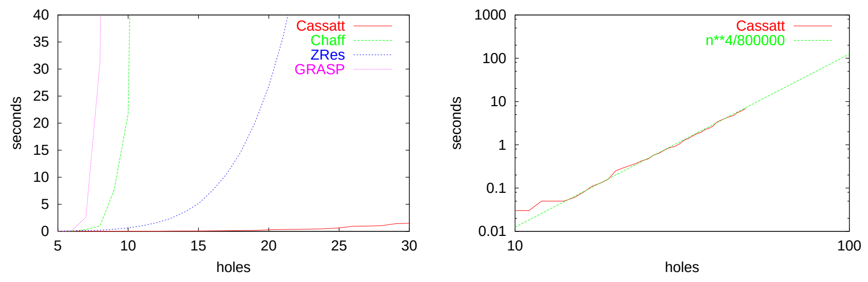

Fig. 7. (a) Runtime of four SAT solvers on hole-n. (b) Cassatt's runtime on hole-n. Cassatt achieves asymptotic speedup over three other solvers

Table 2. Benchmark times and completion ratios for Urquhart-n. Timeouts were set to 500s

Urq-n	Average		Cassatt		Chaff [16]		ZRes [5]		Grasp [17]	
	# Vars	# Clauses	% Solved	Avg	% Solved	Avg	% Solved	Avg	% Solved	Avg
3	44.6	421.2	100%	.07s	50%	139s	100%	.20s	0%	-
4	80.2	781.4	100%	.39s	0%	-	100%	.57s	0%	-
5	124.4	1214.4	100%	1.32s	0%	-	100%	1.35s	0%	-
6	177.4	1633.6	100%	4.33s	0%	-	100%	2.83s	0%	-
7	241.8	2261.4	100%	43.27s	0%	-	100%	5.39s	0%	-
8	315.2	2936.6	90%	43.40s	0%	-	100%	9.20s	0%	-

7 Conclusions

In this paper, we have introduced the Cassatt algorithm, a compressed BFS which solves the SAT decision problem. It leverages the compression power of ZDDs to compact state representation. Also, the novel method for encoding the state uses subsuming semantics of ZDDs to reduce redundancies in the search. In addition, the runtime of this algorithm is dependent on the size of the compressed, implicit representation rather than on the size of an explicit representation. This is the first work to attempt using a compressed BFS as a method for solving the SAT decision problem. Our empirical results show that Cassatt is able to outperform DP and DLL based solvers in some cases. We show this by examining runtimes for Cassatt and comparing this to proven lower bounds for DP and DLL based procedures.

Our ongoing work aims to add Boolean Constraint Propagation to the Cassatt algorithm. With Constraint Propagation, any unsatisfied clause which has only one remaining unassigned variable forces the assignment of that variable. It is hoped that taking this into account will give a reduction in memory requirements and runtime. Another direction for future research is studying the effects of SAT variable and BDD orderings on the performance of the proposed algorithm.

References

1. F. A. Aloul, I. L. Markov, and K. A. Sakallah. Faster SAT and Smaller BDDs via Common Function Structure. *Proc. Intl. Conf. Computer-Aided Design*, 2001.
2. F. A. Aloul, M. Mneimneh, and K. A. Sakallah. Backtrack Search Using ZBDDs. *Intl. Workshop on Logic and Synthesis, (IWLS)*, 2001.
3. P. Beame and R. Karp. The efficiency of resolution and Davis-Putnam procedures . submitted for publication.
4. P. Chatalic and L. Simon. Multi-Resolution on Compressed Sets of Clauses. *Proc. of 12th International Conference on Tools with Artificial Intelligence (ICTAI-2000)*, November 2000.
5. P. Chatalic and L. Simon. ZRes: the old DP meets ZBDDs. *Proc. of the 17th Conf. of Autom. Deduction (CADE)*, 2000.
6. M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem Proving. *Comm. ACM*, 5:394–397, 1962.
7. M. Davis and H. Putnam. A computing procedure for quantification theory. *Jounal of the ACM*, 7:201–215, 1960.
8. P. Ferragina and G. Manzini. An experimental study of a compressed index. *Proc. 12th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2001.
9. R. Gasser. *Harnessing Computational Resources for Efficient Exhaustive Search*. PhD thesis, Swiss Fed Inst Tech, Zurich, 1994.
10. R. L. Graham, M. Grötschel, and L. Lovász, editors. *Handbook of Combinatorics*. MIT Press, January 1996.
11. J. F. Groote and H. Zantema. Resolution and binary decision diagrams cannot simulate each other polynomially. Technical Report UU-CS-2000-14, Utrecht University, 2000.
12. A. San Miguel. Random 3-SAT and BDDs: The Plot Thickens Further . *CP*, 2001.
13. S. Minato. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. *30th ACM/IEEE DAC*, 1993.
14. A. Mishchenko. An Introduction to Zero-Suppressed Binary Decision Diagrams. http://www.ee.pdx.edu/˜alanmi/research/.
15. A. Mishchenko. EXTRA v. 1.3: Software Library Extending CUDD Package: Release 2.3.x. http://www.ee.pdx.edu/˜alanmi/research/extra.htm.
16. M. Moskewicz et al. Chaff: Engineering an Efficient SAT Solver . *Proc. of IEEE/ACM DAC*, pages 530–535, 2001.
17. J. P. Marques Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. *ICCAD* , 1996.
18. F. Somenzi. CUDD: CU Decision Diagram Package Release 2.3.1. http://vlsi.colorado.edu/˜fabio/CUDD/cuddIntro.html.
19. T. E. Uribe and M. E. Stickel. Ordered binary decision diagrams and the Davis-Putnam procedure. In J. P. Jouannaud, editor, *1st Intl. Conf. on Constraints in Comp. Logics*, volume 845 of *LNCS*, pages 34–49. Springer, September 1994.
20. A. Urquhart. Hard examples for resolution. *Journal of the ACM*, 34, 1987.